Lecture 4: Recursion and Sorting I

SEHH2239 Data Structures

Learning Objectives:

- To read, write and apply ***recursion methods***
- To implement sorting methods of ***insertion sorts***, ***bubble sorts*** and ***selection sorts***
- To estimate the ***time efficiency*** of the above sorting methods

What is Recursion?

- Recursion is a problem-solving process that breaks a problem into *identical* but *smaller* problems.
- A method that calls itself is a ***recursive method***. The invocation is a ***recursive call.***
- ***Recursion vs Iteration***
  – A recursion method calls itself.
  – An iteration method contains a loop.
    - E.g. For loop, While loop.

A Simple Example of Recursion

- Classic example--the factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n - 1) \cdot n$$

- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot f(n-1) & else \end{cases}$$

As a Python method:

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

Designing Linear Recursion

1. Base case: we must always have some base cases, which can be solved with recursion

   `e.g. n == 0`

2. Making progress: for the cases that are to be solved recursively, the recursive call must always be to a cast that makes progress toward a base case

   `e.g. n - 1`

- Test for base cases
  - Begin by testing for a set of base cases (there should be *at least one*).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

Tracing a Recursive Method

```
def calcFactorial(n):
    if n == 1:
        return 1
    else:
        return n * calcFactorial(n - 1)
```

```
def calcFactorial(n):
    if n == 1:
        return 1
    else:
        return n * calcFactorial(n - 1)
```

3 * 2 * 1

```
def calcFactorial(n):
    if n == 1:
        return 1
    else:
        return n * calcFactorial(n - 1)
```

3

3 * (2 * 1)

```
def calcFactorial(n):
    if n == 1:
        return 1
    else:
        return n * calcFactorial(n - 1)
```

2

2 * (1)

```
def calcFactorial(n):
    if n == 1:
        return 1
    else:
        return n * calcFactorial(n - 1)
```

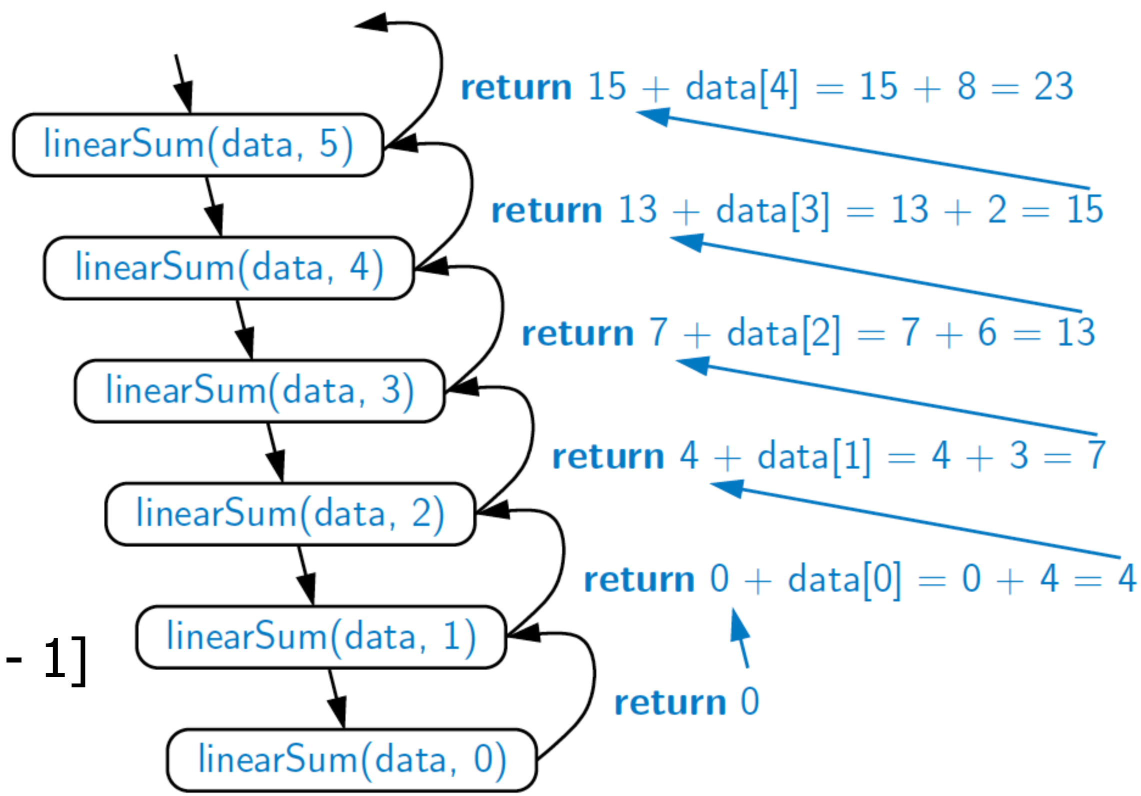
1

Example of Linear Recursion (for further reading)

Algorithm linearSum(A, n):
Input:
  Array, A, of integers
  Integer n such that
    $0 \le n \le |A|$
Output:
    Sum of the first n integers in A

```
if n = 0 then
  return 0
else
  return
linearSum(A, n - 1) + A[n - 1]
```

Recursion trace of linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]



Reversing an Array (for further reading)

Algorithm reverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

```
if i < j then
    Swap A[i] and A[ j]
    reverseArray(A, i + 1, j – 1)
return
```

Comparing data items

Equality: “is” and “==“

- Both “is” and “==” are used for object comparison in Python.
- “==” compares values of two objects,
- “is” checks if two objects are same
  - i.e. two references to same object.

```
a = [1, 2, 3]
b = [1, 2, 3]

print(id(a))
print(id(b))

# "is" operator
if a is b:
    print("Same objects")
else:
    print("Different objects")
```

140031399513936
140031399518448

Different objects

“is” and “==“

- Further Examples

```
# "==" operator
if  a == b:
    print("Same value")
else:
    print("Not the same value")

c = b
if c is b:
    print("Same objects")
else:
    print("Different objects")

d = list(a)
print(d is a)
print(d == a)
```

Same value

Same objects

False
True

Comparing Objects in Class

Object Equal

- Python automatically calls the __eq__ method of a class when you use the == operator to compare the instances of the class.

```
class Student:
  def __init__(self, name, age, mark):
    self.name = name
    self.age = age
    self.dsa_score = mark

  def __eq__(self, other):
    return self.dsa_score == other.dsa_score

joe = Student("Joe Elipse", 20, 80)
pally = Student("Pally Kueng", 30, 75)
martin = Student("Martin Ip", 25, 75)

print(joe == pally)
print(pally == martin)
```

```
False
True
```

Overloaded behaviour of operators

- Python has magic methods to define overloaded behaviour of operators.
- The comparison operators (<, <=, >, >=, == and !=) can be overloaded by providing definition to __lt__, __le__, __gt__, __ge__, __eq__ and __ne__ magic methods

```
class Student2:
  def __init__(self, name, age, mark):
    self.name = name
    self.age = age
    self.dsa_score = mark

  def __eq__(self, other):
    if isinstance(other, Student2):
      return self.dsa_score == other.dsa_score

  def __lt__(self, other):
    if isinstance(other, Student2):
      return self.dsa_score < other.dsa_score

  def __le__(self, other):
    if isinstance(other, Student2):
      return self.dsa_score <= other.dsa_score
```

Click to add text

```
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
operator.__gt__(a, b)
```

Overloaded behaviour of operators

```
joey = Student2("Joey See", 20, 80)
cally = Student2("Cally Peng", 30, 75)
tom = Student2("Tom Miu", 25, 75)
```

```
print(joey == cally)        False
print(cally == tom)         True
print(joey < cally)         False
print(cally < tom)          False

print(joey <= cally)        False
print(cally <= tom)         True
```

SORTING I

What is Sorting?

- Sorting means to put data in order.
- **Ascending** from lowest to highest
- **Descending** from highest to lowest

- E.g. Rearrange a[0], a[1], …, a[n-1] into ascending order. When done, a[0] <= a[1] <= … <= a[n-1]
- 8, 6, 9, 4, 3 => 3, 4, 6, 8, 9

Sorting Algorithms

- A sorting algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements.
- Simple Sorting Algorithms:
  - Insertion Sort
  - Selection Sort
  - Bubble Sort

Insertion Sort

Algorithm of insertion sort

For each a[i] in an array *a* of size *n*,

- Compare a[i] with all elements in the sorted sub-list
- Shift all the elements in the sorted sub-list that > a[i]
- Insert a[i] to the place

Insert An Element

<u>**Insert**</u>

- Given a sorted list/sequence, insert a new element
- E.g. Given 3, 6, 9, 14 and Insert 5
  - Result 3, 5, 6, 9, 14

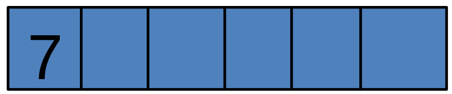
Steps:

Compare new element (5) and last one (14)

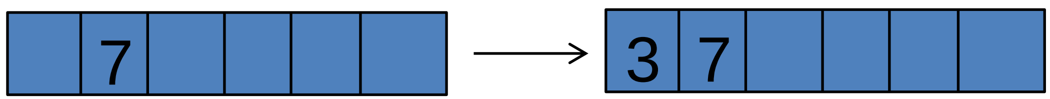
- Shift 14 right to get 3, 6, 9, , 14
- Shift 9 right to get 3, 6, , 9, 14
- Shift 6 right to get 3, , 6, 9, 14
- Insert 5 to get 3, 5, 6, 9, 14

Insertion Sort

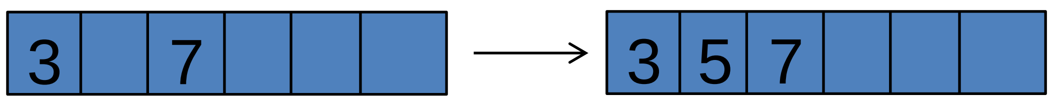
- Sort 7, 3, 5, 6, 1, 8



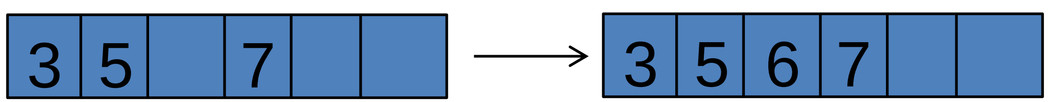
- Start with 7 and insert 3 => 3, 7



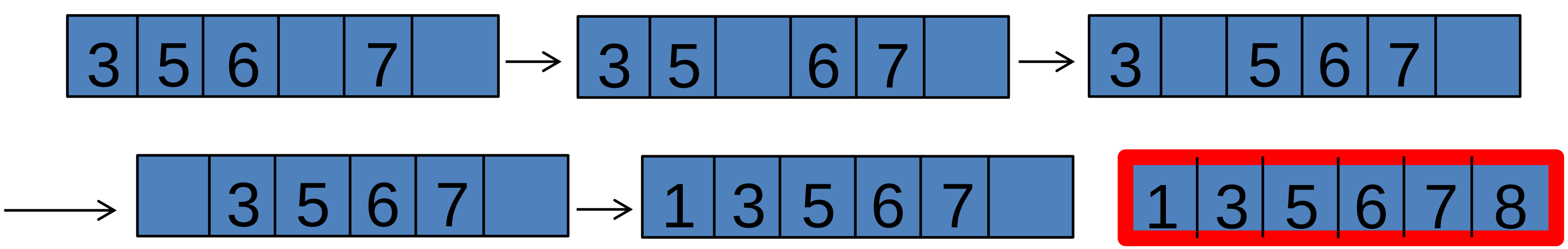
- Insert 5 => 3, 5, 7



- Insert 6 => 3, 5, 6, 7



- Insert 1 => 1, 3, 5, 6, 7



Insertion Sort in Python

```
def insertionSort(arr):
   # Traverse through 1 to len(arr)
   for i in range(1, len(arr)):

      key = arr[i]

      # Move elements of arr[0..i-1], that are
      # greater than key, to one position ahead
      # of their current position
      j = i-1
      while j >= 0 and key < arr[j] :
            arr[j + 1] = arr[j]
            j -= 1
      arr[j + 1] = key

# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):

   print ("% d" % arr[i])
```

This code is contributed by Mohit Kumra

BUBBLE SORT

Algorithm of Bubble Sort

1. In an array *a* of size *n*, if *a*[*i*-1] > *a*[*i*], swap *a*[*i*-1] with *a*[*i*] for *i*=1, 2, 3, … *n*-1. This is called a Bubbling Pass
2. After each bubbling pass, the largest element moves to the right-most position
3. Then, perform bubbling pass for the remaining *n*-1 elements, *a*[0] … *a*[*n*-2], and obtain the largest at *a*[*n*-2]
4. And so on until performing bubbling pass for the remaining 2 elements and obtaining the largest at *a*[1]

Bubble Sort

First Bubbling Pass

- Sort 6, 5, 8, 4, 3, 1
- 6 > 5, swap 6 and 5 => 5, 6, 8, 4, 3, 1
- 6 < 8, no swap => 5, 6, 8, 4, 3, 1
- 8 > 4, swap 8 and 4 => 5, 6, 4, 8, 3, 1
- 8 > 3, swap 8 and 3 => 5, 6, 4, 3, 8, 1
- 8 > 1, swap 8 and 1 => 5, 6, 4, 3, 1, 8
- After the **1st pass**, the largest element 8 moves to the right-most position

Bubble Sort

Second Bubbling Pass

- Sort the remaining 5, 6, 4, 3, 1
- 5 < 6, no swap => 5, 6, 4, 3, 1
- 6 > 4, swap 6 and 4 => 5, 4, 6, 3, 1
- 6 > 3, swap 6 and 3 => 5, 4, 3, 6, 1
- 6 > 1, swap 6 and 1 => 5, 4, 3, 1, 6
- After the **2nd pass**, the largest element 6 moves to the right-most position

Bubble Sort

Keep on Bubbling

- After the **3rd pass**, the largest element 5 moves to the right-most position, and the result is: 4, 3, 1, 5
- After the **4th pass**, the result is: 3, 1, 4
- After the **5th pass**, the result is: 1, 3
- At the **6th pass**, one element remains, that is, 1
- Combining *a*[0]=1, *a*[1]=3, … *a*[5]=8 to get the sorted array 1, 3, 4, 5, 6, 8.

Bubble Sort

```python
def bubble_sort(our_list):
    # We go through the list as many times as there are elements
    for i in range(len(our_list)):
        # We want the last pair of adjacent elements to be (n-2, n-1)
        for j in range(len(our_list) - 1):
            if our_list[j] > our_list[j+1]:
                # Swap
                our_list[j], our_list[j+1] = our_list[j+1], our_list[j]


our_list = [19, 13, 6, 2, 18, 8]
bubble_sort(our_list)
print(our_list)
```

#https://stackabuse.com/bubble-sort-in-python/

Early-Terminating Bubble Sort

Bubble Sort

- **Sort 6, 1, 2, 5, 3, 4**
- **1st pass => 1, 2, 5, 3, 4, 6**
- **2nd pass => 1, 2, 3, 4, 5, 6**
- **3rd pass, no swap => 1, 2, 3, 4, 5, 6**
- **4th pass, no swap => 1, 2, 3, 4, 5, 6**
- **5th pass, no swap => 1, 2, 3, 4, 5, 6**
- **6th pass, no swap => 1, 2, 3, 4, 5, 6**
- **Terminate and get the sorted list**

No need after this point

Early-Terminating Bubble Sort

- **Sort 6, 1, 2, 5, 3, 4**
- **1st pass => 1, 2, 5, 3, 4, 6**
- **2nd pass => 1, 2, 3, 4, 5, 6**
- **3rd pass, no swap => 1, 2, 3, 4, 5, 6**
- **Terminate and get sorted list**

Early-Terminating Bubble Sort

- If a bubbling pass results in no swaps, then the array is in sorted order and no further bubbling passes are necessary.
- *Note: It needs a pass for the checking of order*

Early-Terminating Bubble Sort

```
def bubble_sort_earlyterm(our_list):
    has_swapped = True

    num_of_iterations = 0

    while(has_swapped):
        has_swapped = False
        for i in range(len(our_list) - num_of_iterations - 1):
            if our_list[i] > our_list[i+1]:
                # Swap
                our_list[i], our_list[i+1] = our_list[i+1], our_list[i]
                has_swapped = True
        num_of_iterations += 1

our_list = [19, 13, 6, 2, 18, 8]
bubble_sort_earlyterm(our_list)
print(our_list)
```

#https://stackabuse.com/bubble-sort-in-python/

SELECTION SORT

Algorithm of Selection Sort

1. In an array *a* of size *n*, determine the largest element and swap the largest with the last element *a*[*n*-1]
2. Then, determine the largest of the remaining *n*-1 elements, *a*[0] … *a*[*n*-2], and swap that largest with *a*[*n*-2]
3. And so on until determining the largest of the remaining 2 elements and swap the largest with *a*[1]

Selection Sort

- Sort 6, 5, 8, 4, 3, 1
- Largest is 8, swap with 1 => 6, 5, 1, 4, 3, 8
- Sort 6, 5, 1, 4, 3
- Largest is 6, swap with 3 => 3, 5, 1, 4, 6, 8
- Sort 3, 5, 1, 4
- Largest is 5, swap with 4 => 3, 4, 1, 5, 6, 8
- Sort 3, 4, 1
- Largest is 4, swap with 1 => 3, 1, 4, 5, 6, 8
- Sort 3, 1
- Largest is 3, swap with 1 => 1, 3, 4, 5, 6, 8

Selection Sort

```python
def selection_sort(L):
    # i indicates how many items were sorted
    for i in range(len(L)-1):
        # To find the minimum value of the unsorted segment
        # We first assume that the first element is the lowest
        min_index = i
        # We then use j to loop through the remaining elements
        for j in range(i+1, len(L)-1):
            # Update the min_index if the element at j is lower than it
            if L[j] < L[min_index]:
                min_index = j
        # After finding the lowest item of the unsorted regions, swap with the first unsorted item
        L[i], L[min_index] = L[min_index], L[i]
```

L = [3, 1, 41, 59, 26, 53, 59] print(L)
selection_sort(L) *# Let's see the list after we run the Selection Sort* print(L)

https://stackabuse.com/selection-sort-in-python/

Early-Terminating Selection Sort

- Shortcoming: iteration keeps on even if the elements have been sorted
- Use Early-Terminating Selection Sort to eliminate unnecessary iterations
- In Early-Terminating Selection Sort, during the scan for the largest element, it checks to see whether the array is already sorted or not

Summary on Key terms

- **Recursion**
- **Insertion sort**
- **Bubble sort**
  - Early-Terminating
- **Selection sort**